Pointers

9/27/16

Addresses in memory

- `float values[100];`
- `4(%ebp)`
- `read_int(&x);`
- `int* unsigned_output;`

Pointers in C

- Like any other variable, must be declared:
  - Using the format: `type *name;`
- Example: `int *myptr;`
  - This is a promise to the compiler:
    - This variable holds a memory address. If you follow what it points to in memory (dereference it), you'll find an integer.
- A note on syntax:
  - `int* myptr;`   `int * myptr;`   `int *myptr;`
  - These all do the same thing. (note the `*` position)

Declaring pointer variables

```
float f;    //declares a float
int i = 0; //declares and initializes an int


float* fp;         //declares a float pointer
int *ip = NULL; //declares and inits an
                //int pointer
```

Pointers store addresses.

```
float f=0, *fp=NULL;
int i=0, *ip=NULL;

printf("%f, %p\n", f, fp);
printf("%d, %p\n", i, ip);
```

```
> 0.000000, 0x0
> 0, 0x0
```

Pointer operators: `*` and `&`

- `*` is the *value-at-address* operator.
  - AKA the *dereference* operator.

- `&` is the *address-of* operator.

```
float f, *fp;
fp = &f;
*fp = 0;
```

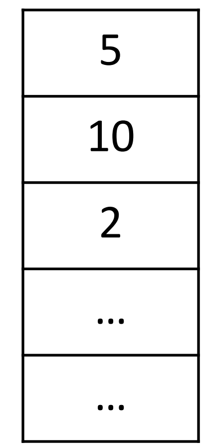
Putting a * in front of a variable…

- When you *declare* the variable:
  - Declares the variable to be a pointer
  - It stores a memory address

- When you *use* the variable (dereference):
  - Like putting () around a register name
  - Follows the pointer out to memory
  - Acts like the specified type (e.g., int, float, etc.)

Suppose we set up a pointer like this. Which expression gives us 5, and which gives us a memory address?

```
int *iptr = ???; //
```



A. Memory address: `*iptr`, Value 5: `iptr`

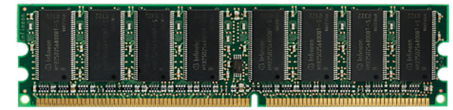
B. Memory address: `iptr`, Value 5: `*iptr`

So we declared a pointer…

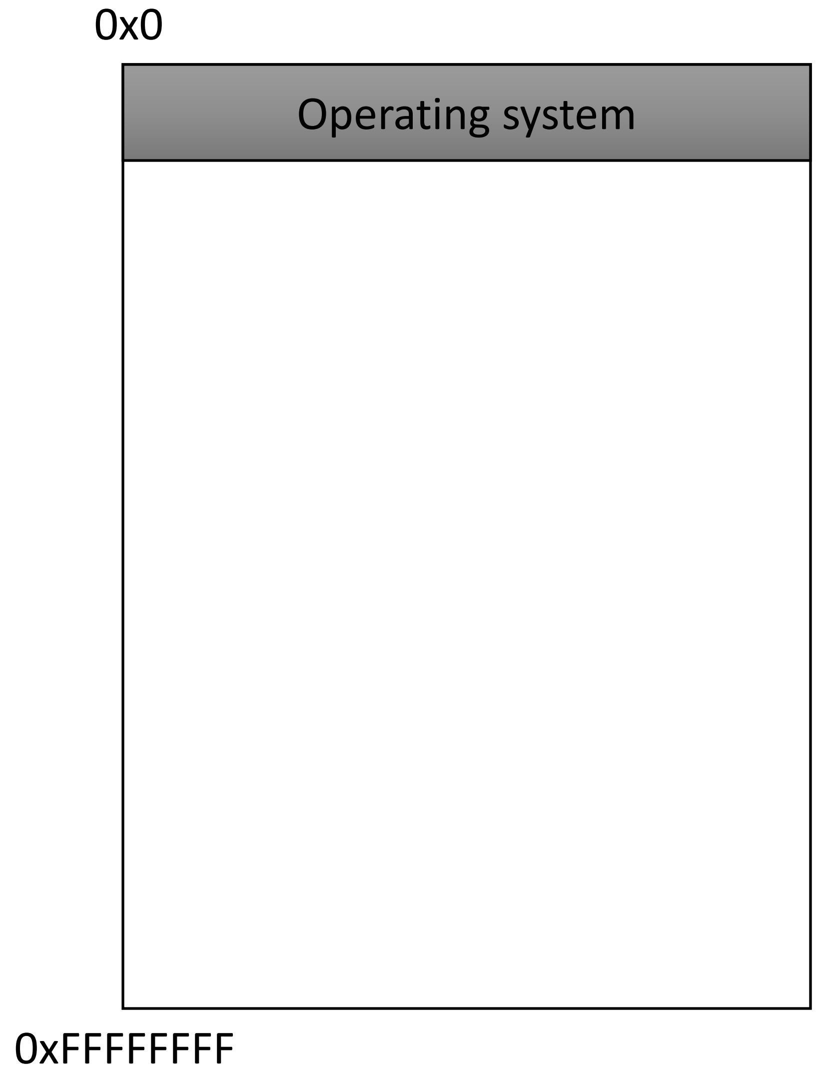
- How do we make it point to something?
    1. Assign it the address of an existing variable
    2. Copy some other pointer
    3. Allocate some memory and point to it

- First, let’s look at how memory is organized.

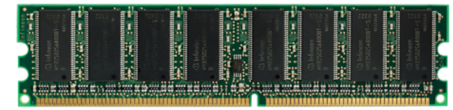
Memory



- Behaves like a big array of bytes, each with an address (bucket #).
- By convention, we divide it into regions.
- The region at the lowest addresses is usually reserved for the OS.

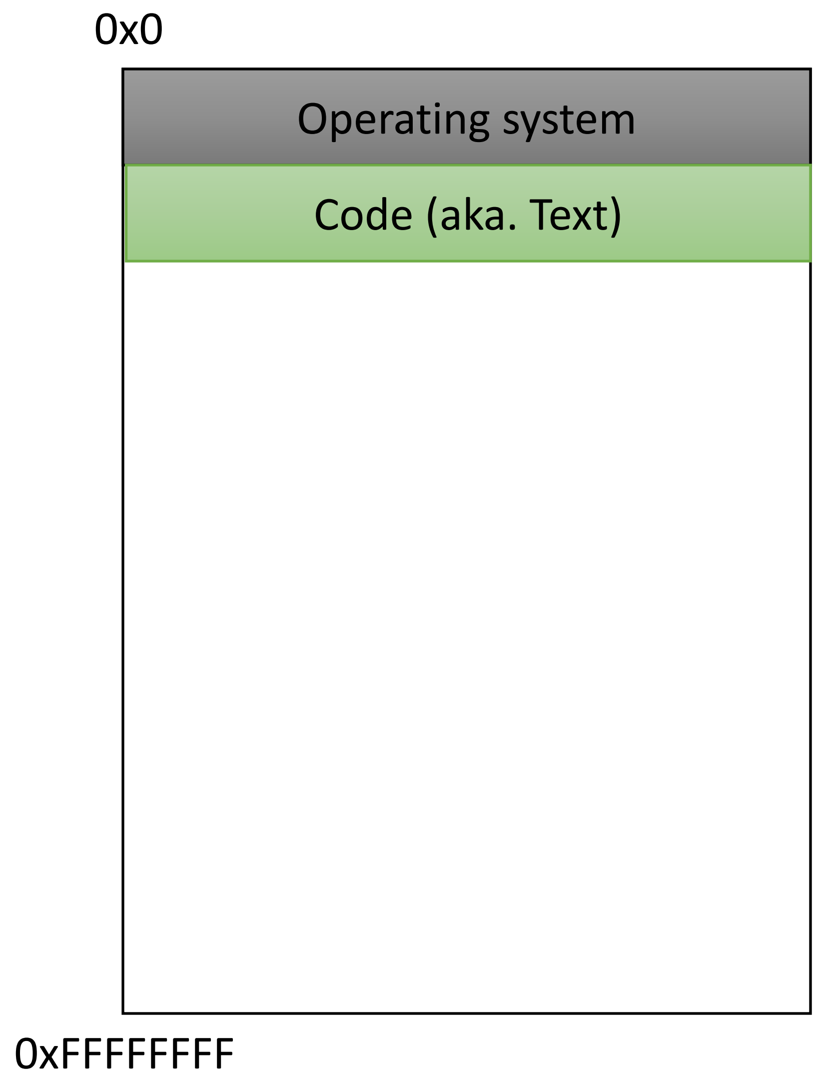


Memory - Text

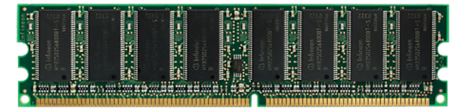


- After the OS, we store the program’s code.

- Instructions generated by the compiler.

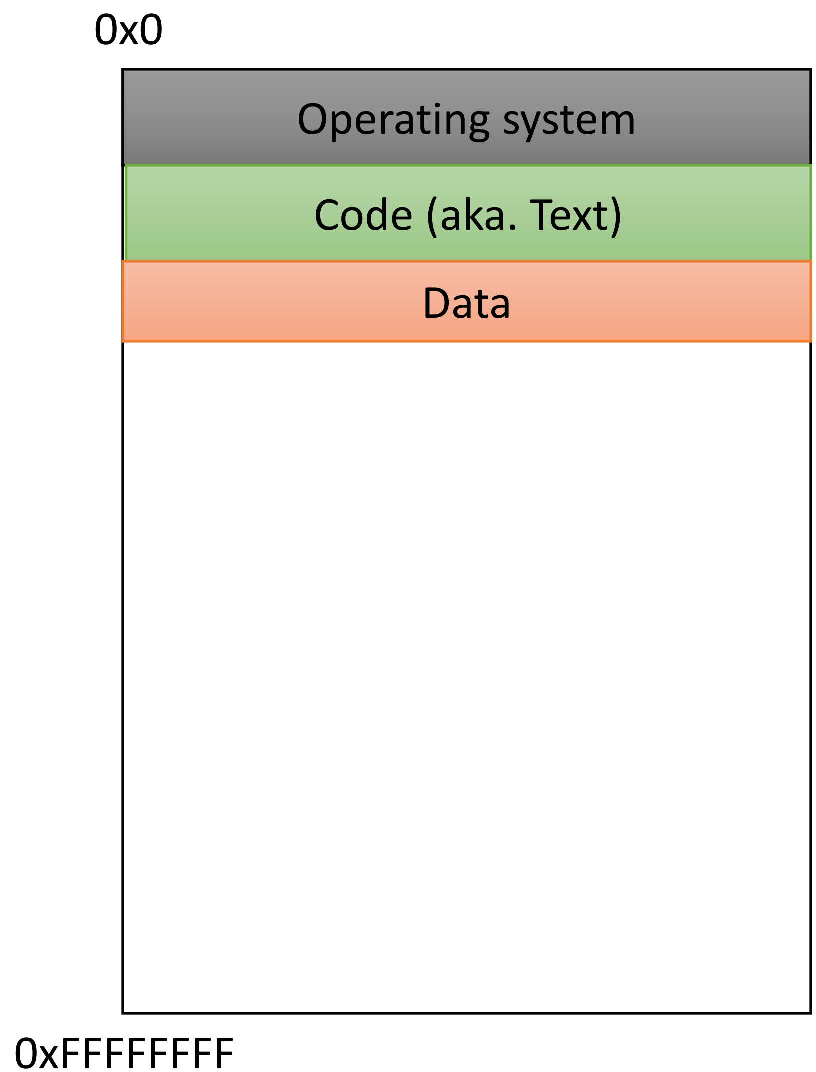


Memory – (Static) Data

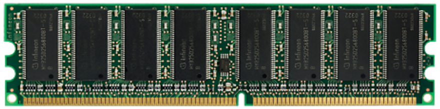


- Next, there's a fixed-size region for static data.

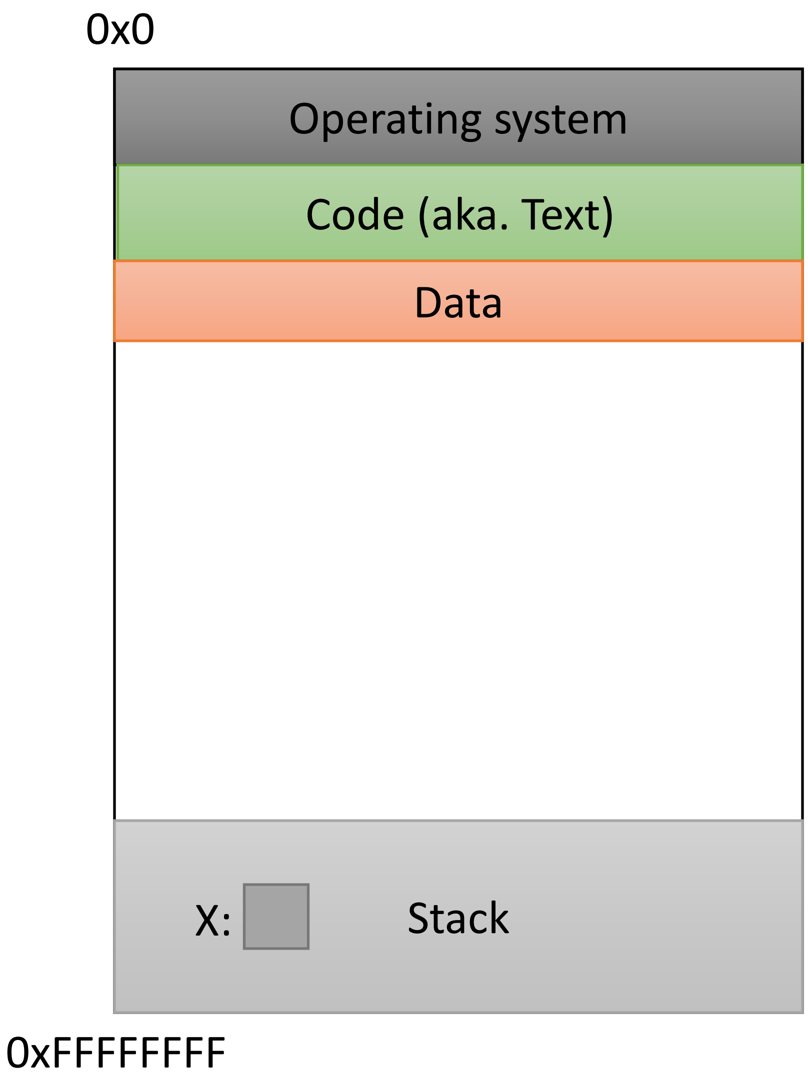
- This stores static variables that are known at compile time.
  - Global variables



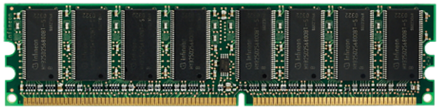
Memory - Stack



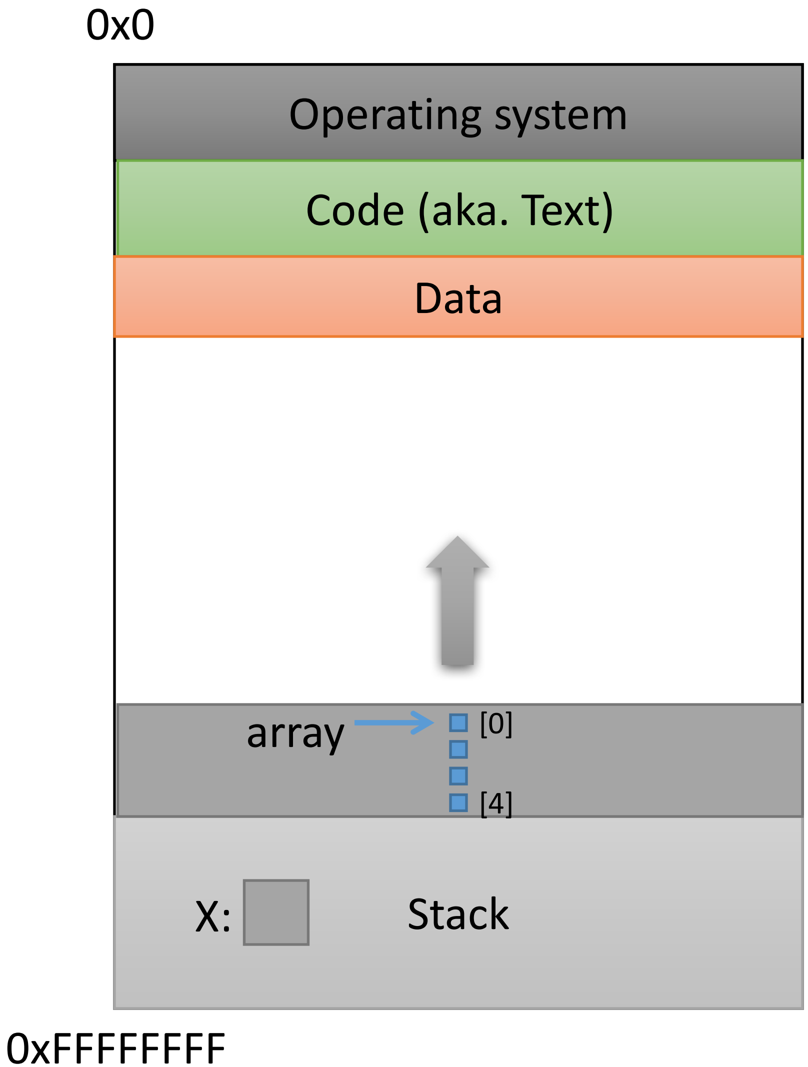
- At high addresses, we keep the stack.
- This stores local (automatic) variables.
  - The kind we've been using in C so far.
  - e.g., int x;



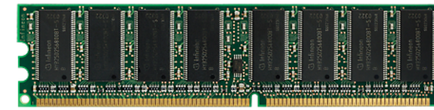
Memory - Stack



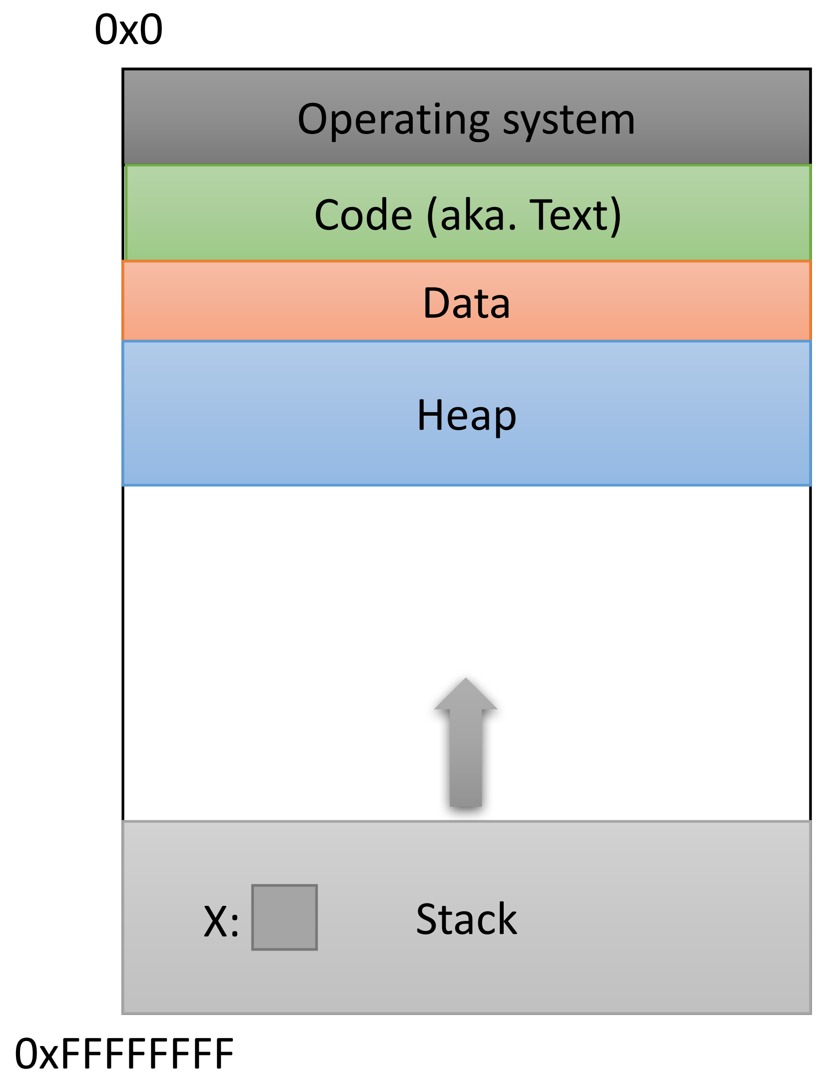
- The stack grows upwards towards lower addresses (negative direction).
- Example: Allocating array
    - int array[4];
- (Note: this differs from Python.)



Memory - Heap



- The heap stores dynamically allocated variables.
- When programs explicitly ask the OS for memory, it comes from the heap.
  - `malloc()` function



If we can declare variables on the stack, why do we need to dynamically allocate things on the heap?

A. There is more space available on the heap.

B. Heap memory is better. (Why?)

C. We may not know a variable’s size in advance.

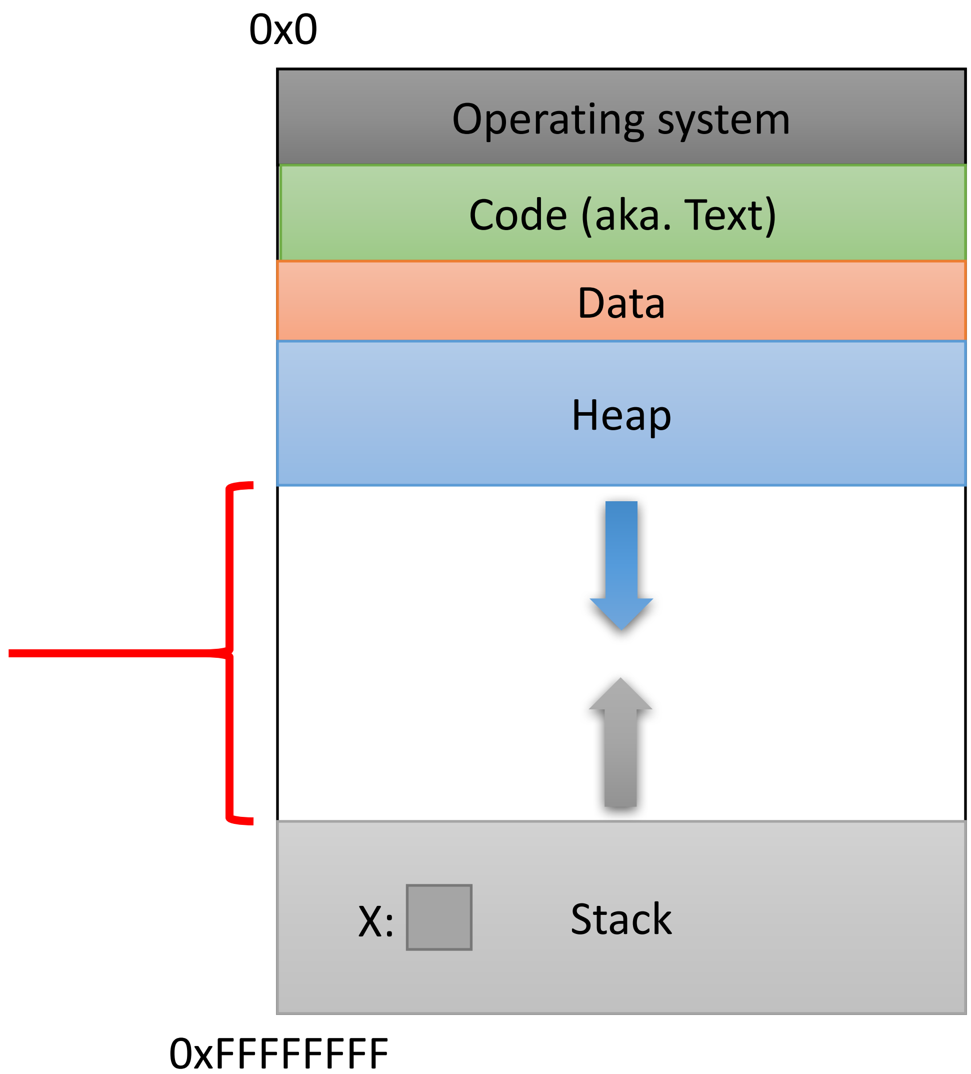
D. The stack grows and shrinks automatically.

E. Some other reason.

Memory - Heap

- The heap grows downwards, towards higher addresses.

- This picture is not to scale – the gap is huge.



Which region would we expect the PC register (%eip) to point to?

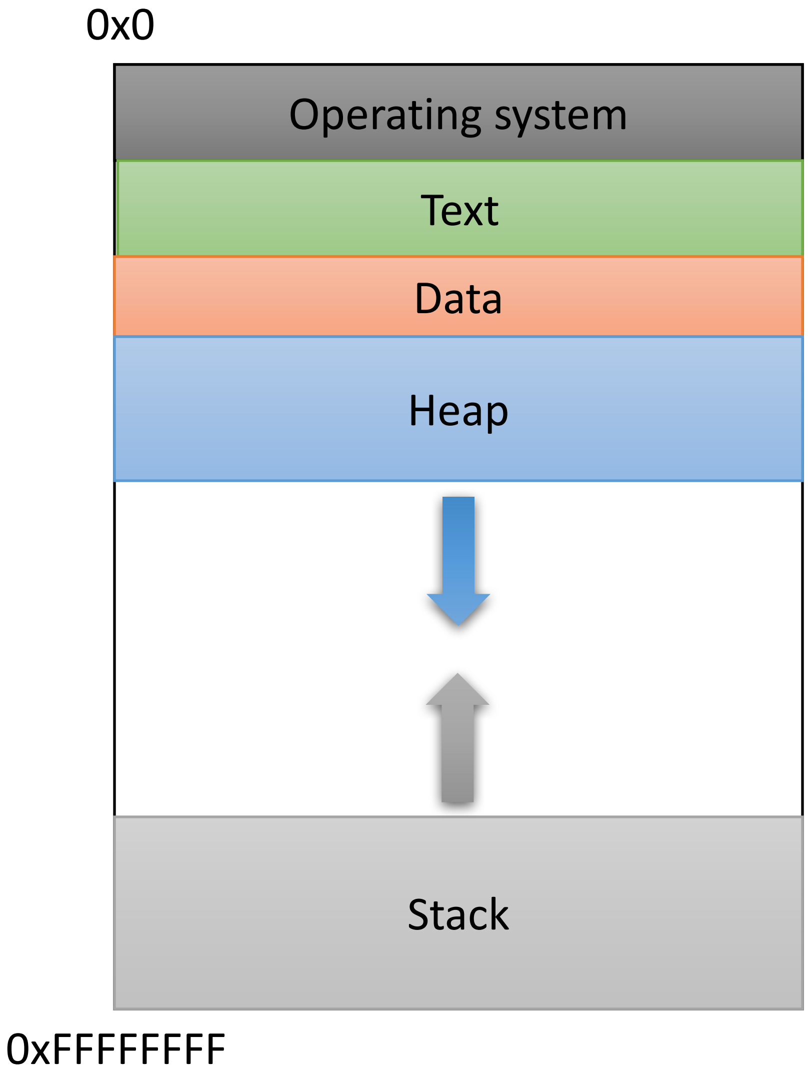
A. OS

B. Text

C. Data

D. Heap

E. Stack



What should happen if we try to access an address that’s NOT in one of these regions?

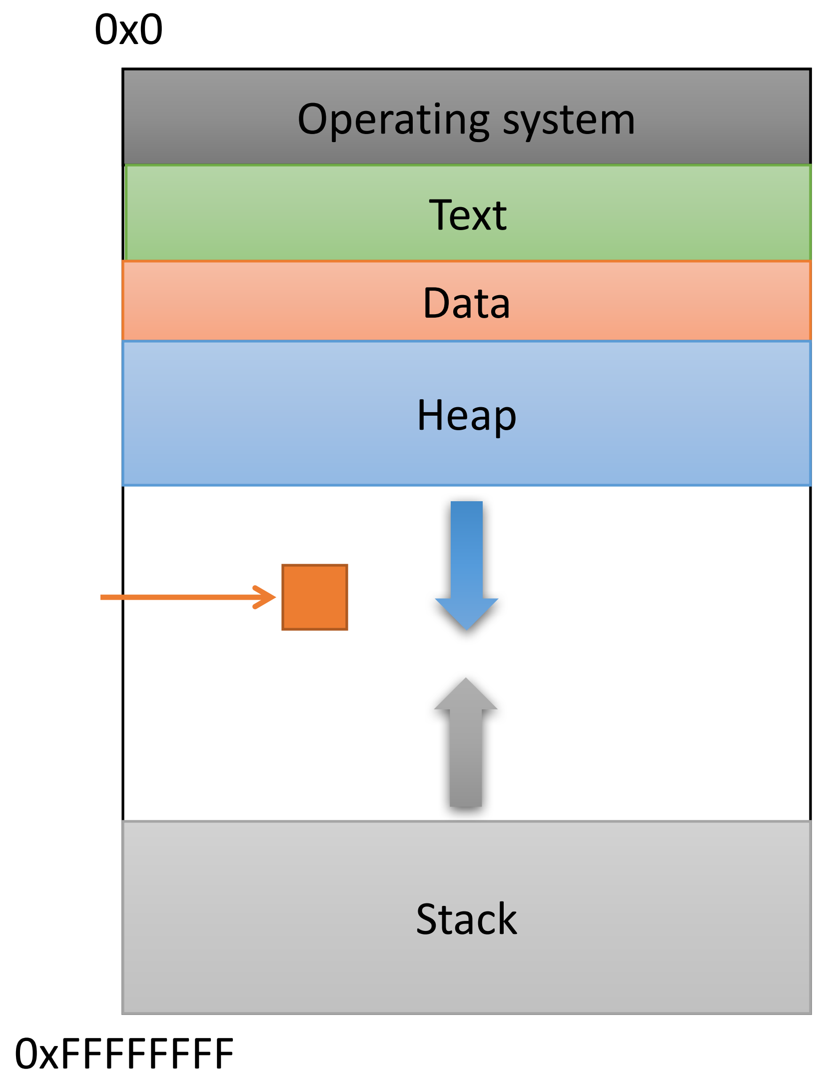
A. The address is allocated to your program.

B. The OS warns your program.

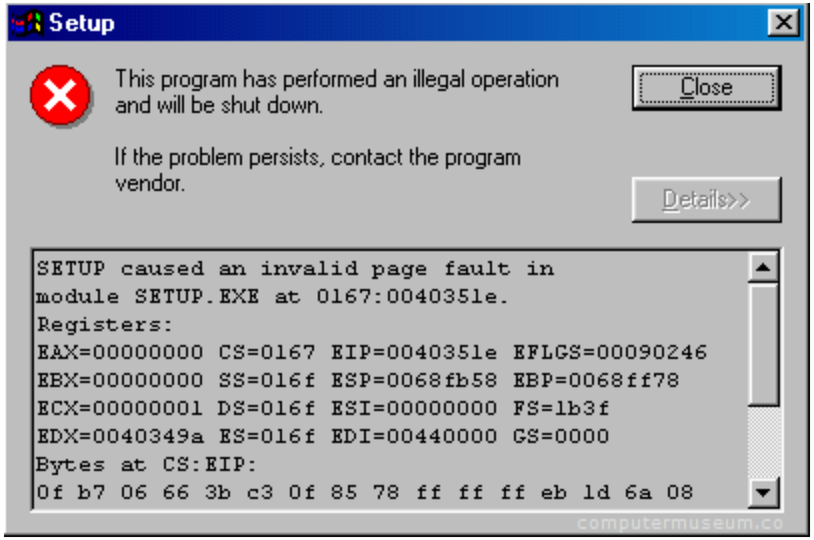
C. The OS kills your program.

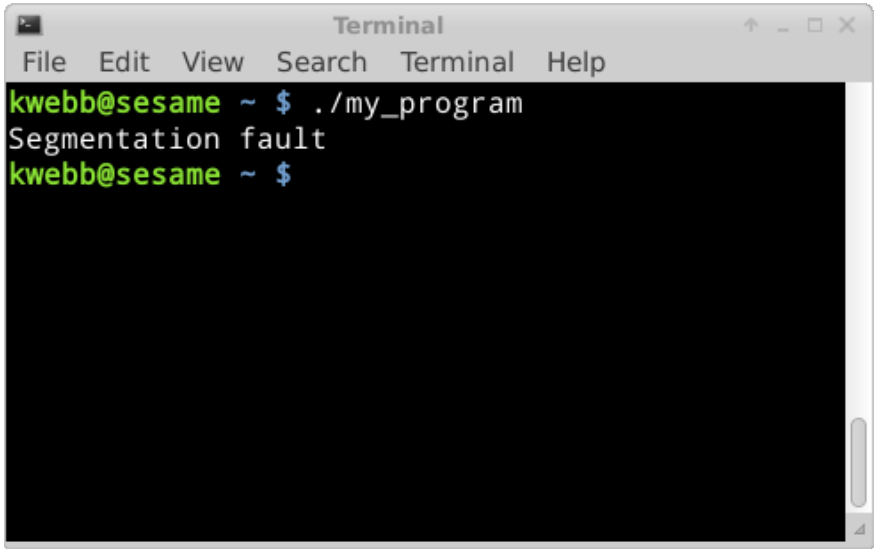
D. The access fails, try the next instruction.

E. Something else



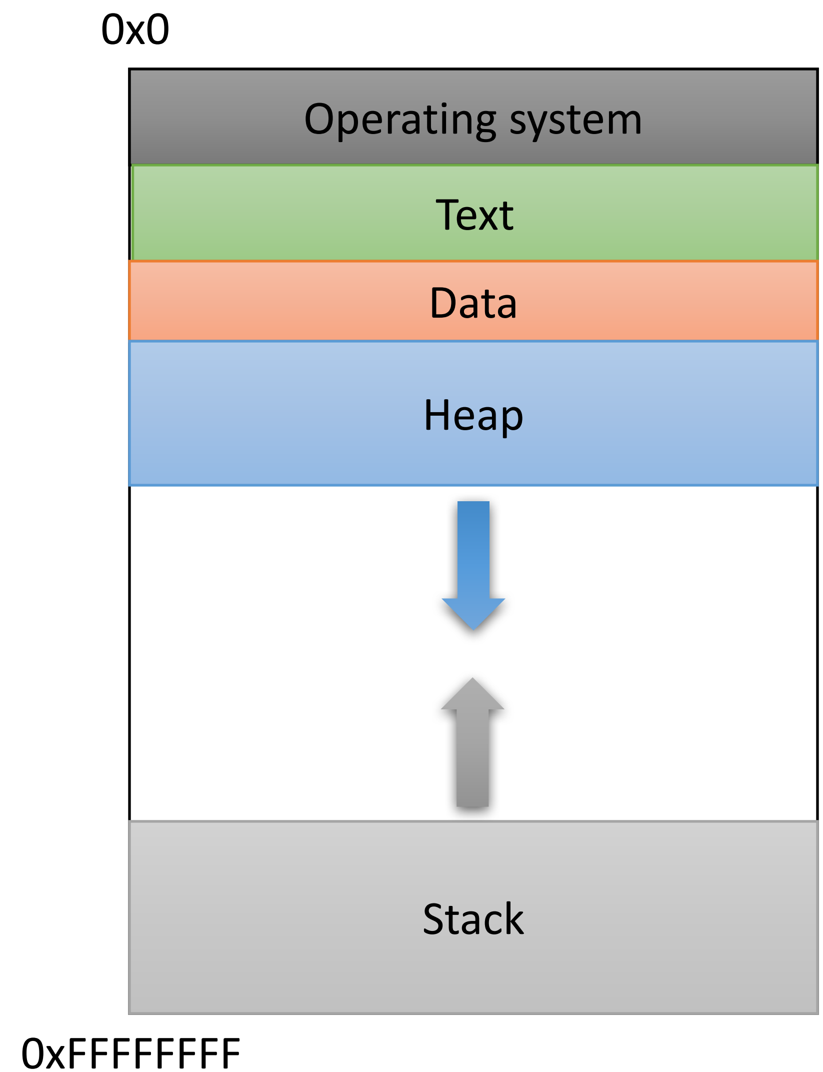
Segmentation Violation





Segmentation Violation

- Each region also known as a memory segment.
- Accessing memory outside a segment is not allowed.
- Can also happen if you try to access a segment in an invalid way.
  - OS not accessible to users
  - Text is usually read-only

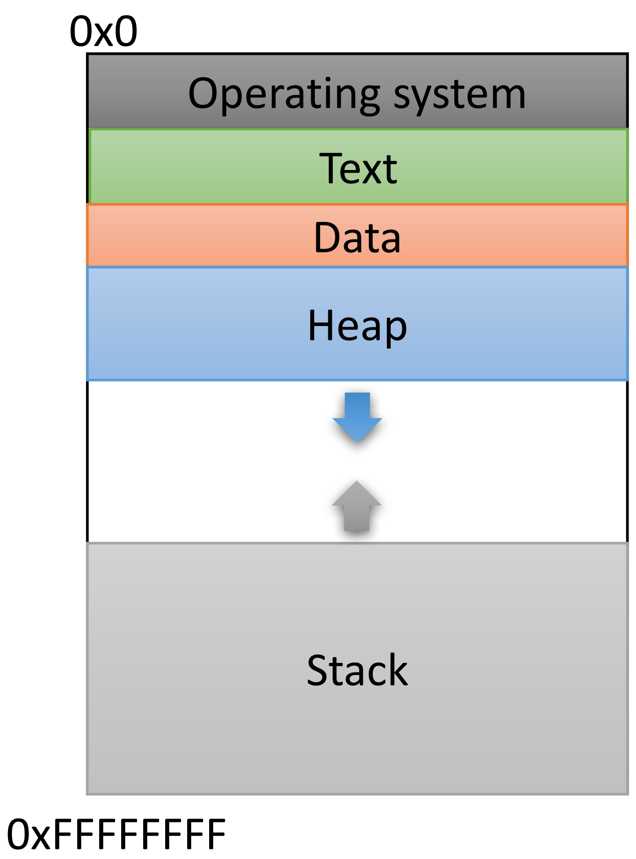


Recap

- & gives us the address of a variable (a pointer)
- * allows us to follow the address to m
  the item (dereference the pointer)

- Memory model:
- So far, all variables on stack.

- Up next: using the heap.
  - We may not know the size of
    a variable in advance. (dynamic)



So we declared a pointer…

- How do we make it point to something?
  1. Assign it the address of an existing variable
  2. Copy some other pointer
  3. Allocate some memory and point to it

The Address Of (`&`)

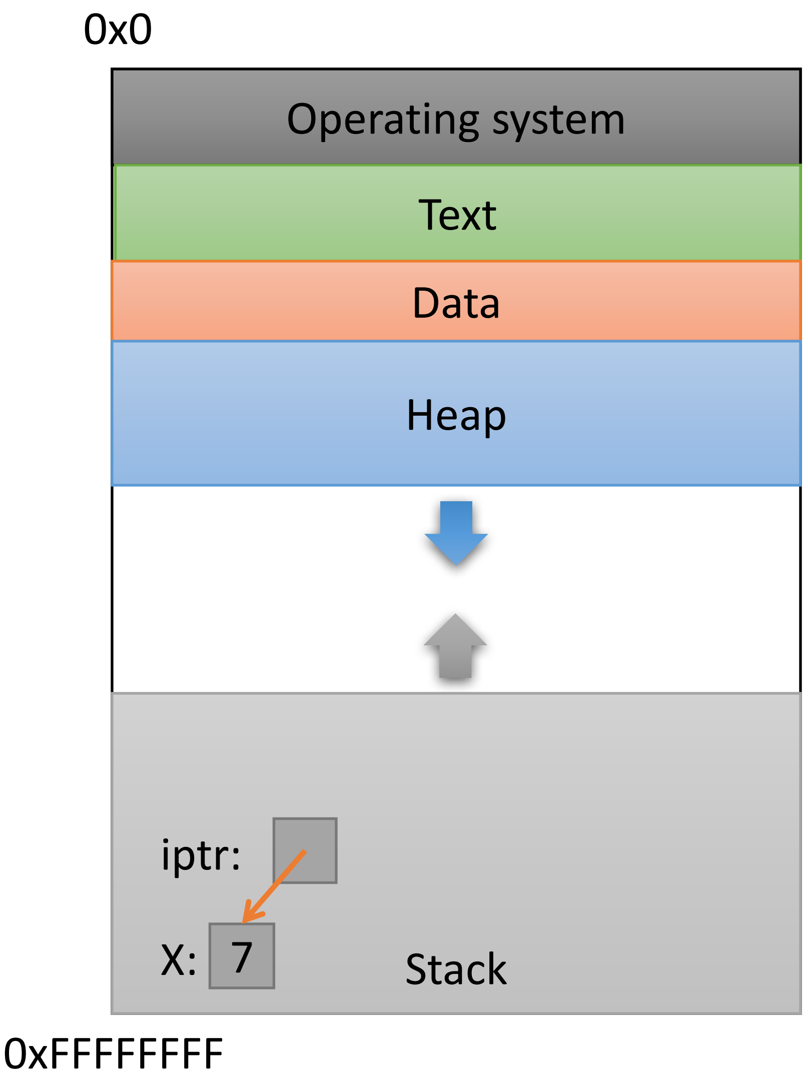
- You can create a pointer to anything by taking its address with the *address of* operator (`&`).

The Address Of (&)

```
int main() {
   int x = 7;
   int *iptr = &x;



   return 0;
}
```

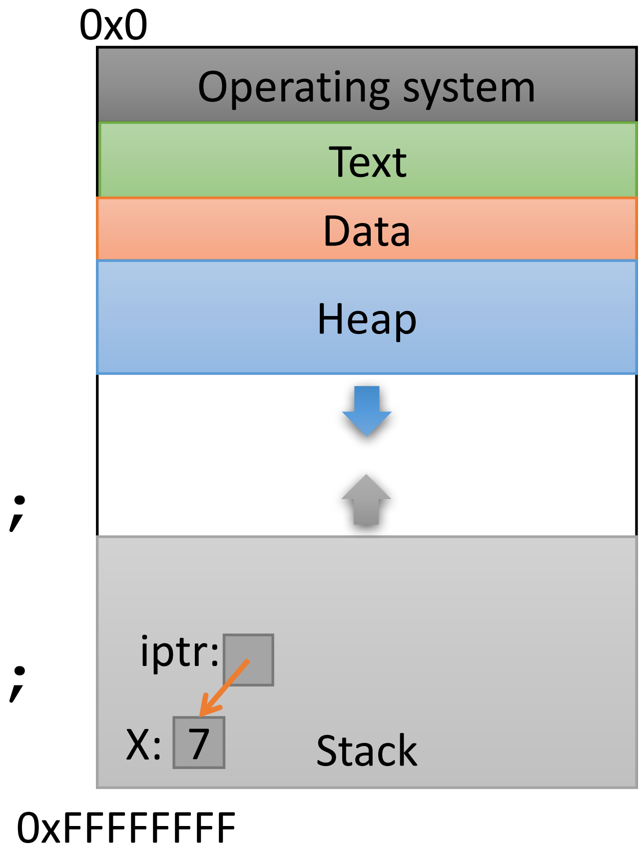


What would this print?

```
int main() {
  int x = 7;
  int *iptr  = &x;
  int *iptr2 = &x;

  printf("%d %d ", x, *iptr);
  *iptr2 = 5;
  printf("%d %d ", x, *iptr);

  return 0;
}
```



A. 7 7 7 7   B. 7 7 7 5   C. 7 7 5 5   D. Something else

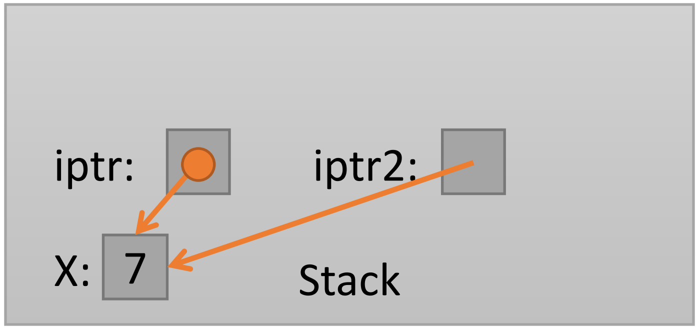
So we declared a pointer…

- How do we make it point to something?
  1. Assign it the address of an existing variable
  2. Copy some other pointer
  3. Allocate some memory and point to it

Copying a Pointer

- We can perform assignment on pointers to copy the stored address.

```
int x = 7;
int *iptr, *iptr2;
iptr = &x;
iptr2 = iptr;
```



Pointer Types

- By default, we can only assign a pointer if the type matches what C expects.

```c
int x = 7;
int *iptr = &x;
```

```c
int x = 7;
float *fptr = &x;
```

- "Warning: initialization from incompatible pointer type" (Don't ignore this!)

`void *`

- There exists a special type, `void *`, which represents "generic pointer" type.
  - Can be assigned to any pointer variable
  - `int *iptr = (void *) &x;`

- This is useful for cases when:
  1. You want to create a generic "safe value" that you can assign to any pointer variable.
  2. You want to pass a pointer to / return a pointer from a function, but you don't know its type.
  3. You know better than the compiler that what you're doing is safe, and you want to eliminate the warning.

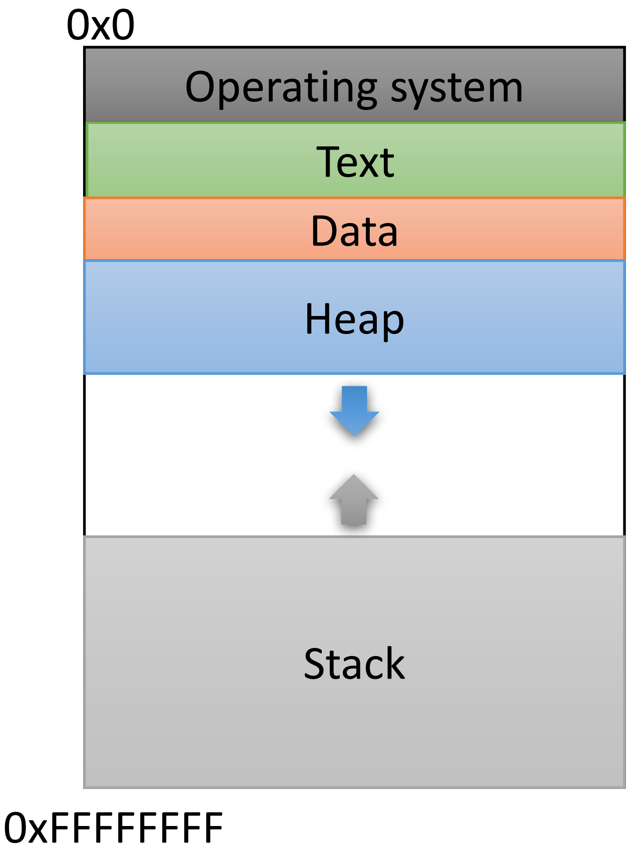
`NULL`: A special pointer value.

- You can assign `NULL` to any pointer, regardless of what type it points to (it's a `void *`).
  - `int *iptr = NULL;`
  - `float *fptr = NULL;`

- `NULL` is equivalent to pointing at memory address `0x0`. This address is NEVER in a valid segment of your program's memory.
  - This guarantees a segfault if you try to deref it.
  - Generally a good ideal to initialize pointers to `NULL`.

What will this do?

```
int main() {
   int *ptr;
   printf(“%d”, *ptr);
}
```



A. Print 0
B. Print a garbage value
C. Segmentation fault
D. Something else

Takeaway: If you’re not immediately assigning it something when you declare it, initialize your pointers to NULL.

So we declared a pointer…

- How do we make it point to something?
  1. Assign it the address of an existing variable
  2. Copy some other pointer
  3. Allocate some memory and point to it

Allocating (Heap) Memory

- The standard C library #include <stdlib.h> includes functions for allocating memory

```
void *malloc(size_t size)
```

- Allocate `size` bytes on the heap and return a pointer to the beginning of the memory block

```
void free(void *ptr)
```

- Release the `malloc`'ed block of memory starting at `ptr` back to the system

Recall: `void *`

- `void *` is a special type that represents "generic pointer".
  - Can be assigned to any pointer variable

- This is useful for cases when:
  1. You want to create a generic "safe value" that you can assign to any pointer variable.
  2. You want to pass a pointer to / return a pointer from a function, but you don't know its type.
  3. You know better than the compiler that what you're doing is safe, and you want to eliminate the warning.

- When `malloc()` gives you bytes, it doesn't know or care what you use them for.

The `sizeof()` operator

`void *malloc(size_t size)`

- Allocate `size` bytes on the heap and return a pointer to the beginning of the memory block

- How much memory should we ask for?

- Use C's `sizeof()` operator:

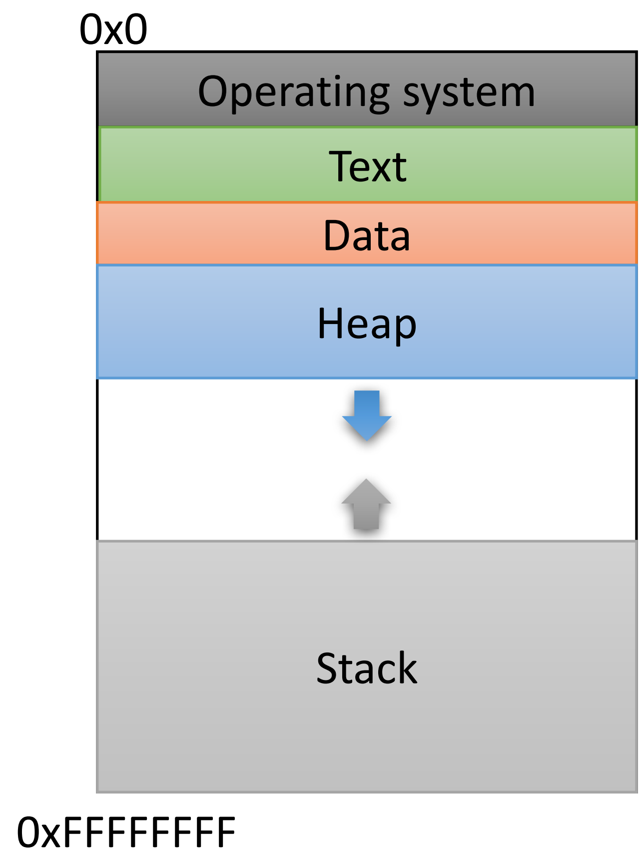
```
int *iptr = NULL;
iptr = malloc(sizeof(int));
```

Example

```
int *iptr = NULL;

iptr = malloc(sizeof(int));

*iptr = 5;
```

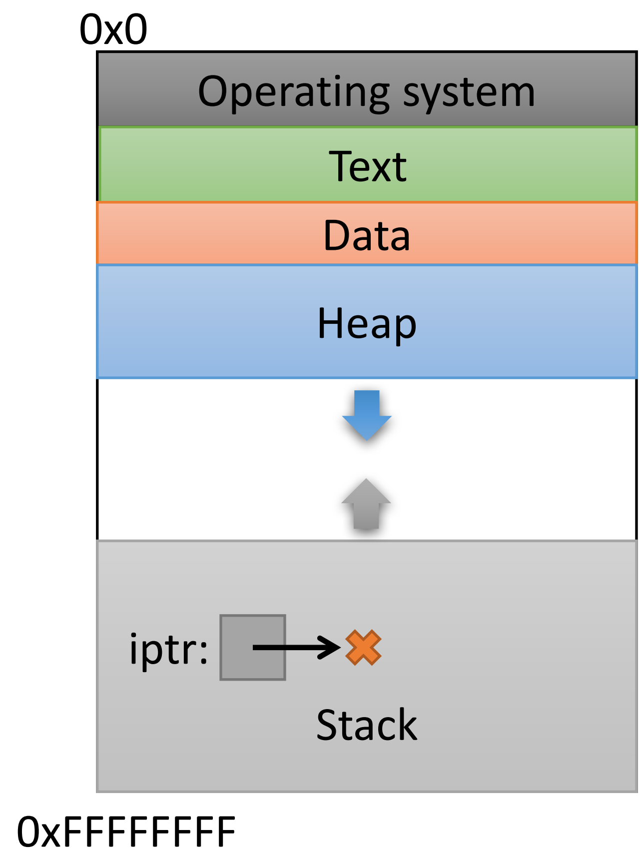


Example

```
int *iptr = NULL;

iptr = malloc(sizeof(int));

*iptr = 5;
```



Create an integer pointer, named iptr, on the stack.

Assign it NULL.

Example

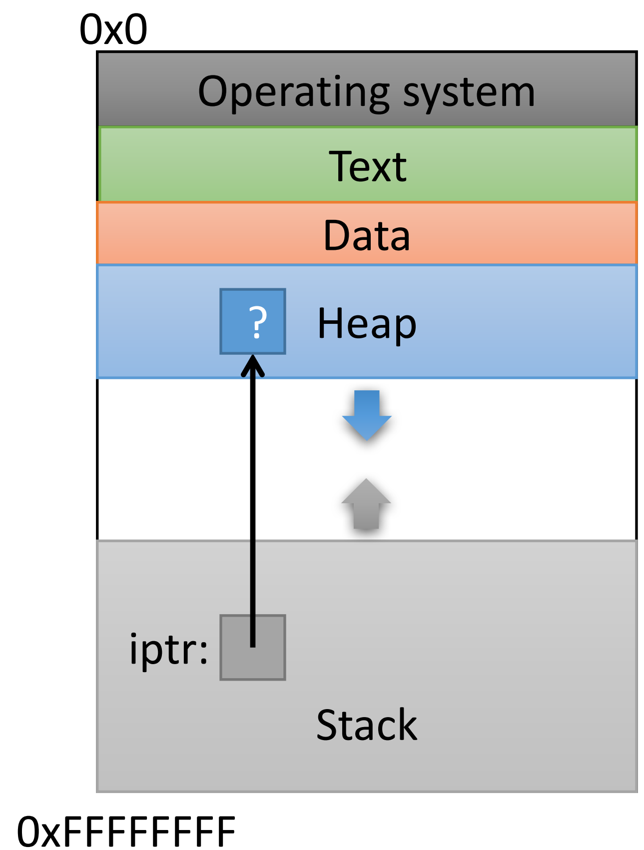
```
int *iptr = NULL;

iptr = malloc(sizeof(int));

*iptr = 5;
```

Allocate space for an integer on the heap (4 bytes), and return a pointer to that space.

Assign that pointer to iptr.



What value is stored in that area right now?

Who knows… Garbage.

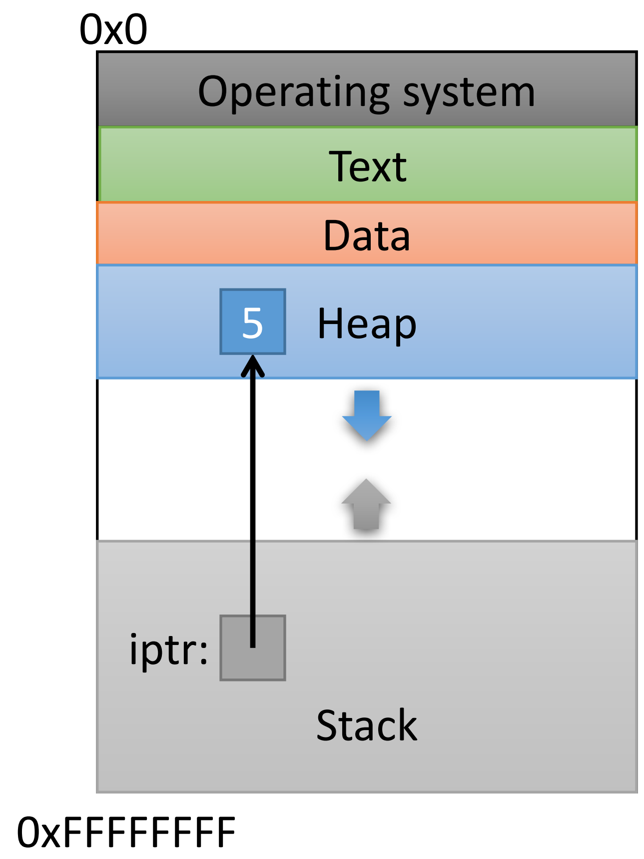
Example

```
int *iptr = NULL;

iptr = malloc(sizeof(int));

*iptr = 5;
```

Use the allocated heap space by dereferencing the pointer.



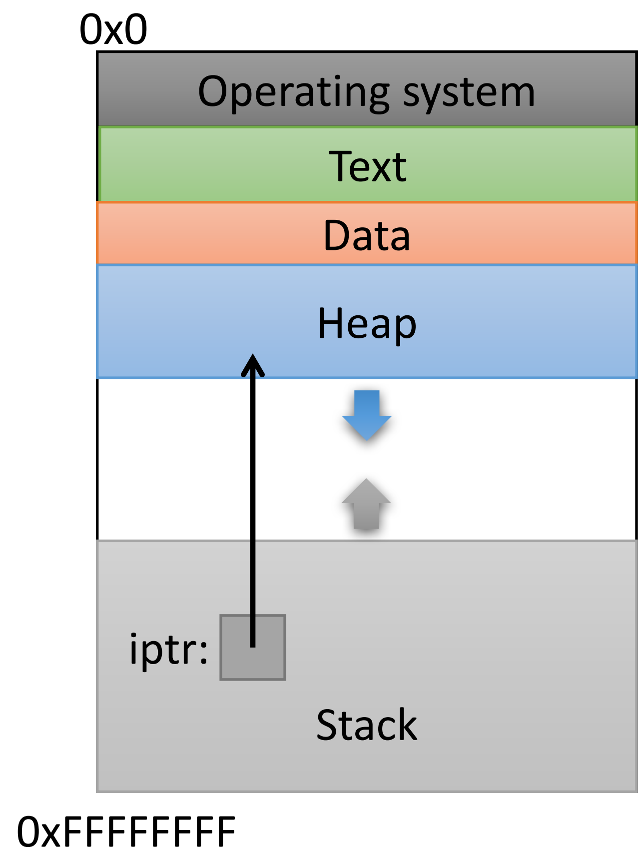
Example

```
int *iptr = NULL;

iptr = malloc(sizeof(int));

*iptr = 5;

free(iptr);
```



Free up the heap memory we used.

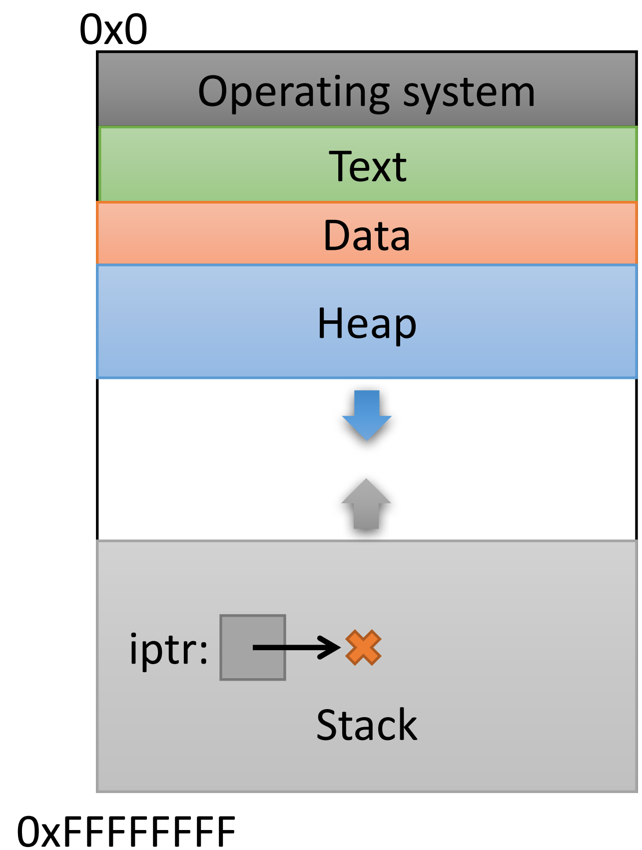
Example

```
int *iptr = NULL;

iptr = malloc(sizeof(int));

*iptr = 5;

free(iptr);
iptr = NULL;
```



Clean up this pointer, since it's no longer valid.

`sizeof()`

- Despite the ()’s, it’s an operator, not a function
  - Other operators:
    - addition / subtraction (+ / -)
    - address of (&)
    - indirection (*) (dereference a pointer)

- Works on any type to tell you how much memory it needs.

`sizeof()` example

```
struct student {
  char name[40];
  int age;
  double gpa;
}
```

How many bytes is this?
Who cares…
Let the compiler figure that out.

```
struct student *bob = NULL;
bob = malloc(sizeof(struct student));
```

I don't ever want to see a number hard-coded in here!

You’re designing a system. What should happen if a program requests memory and the system doesn’t have enough available?

A. The OS kills the requesting program.
B. The OS kills another program to make room.
C. malloc gives it as much memory as is available.
D. malloc returns NULL.
E. Something else.

Running out of Memory

- If you're ever unsure of malloc / free's behavior:

```
$ man malloc
```

- According to the C standard:

"The malloc() function returns a pointer to the allocated memory that is suitably aligned for any kind of variable. **On error, this function returns NULL.**"

- Further down in the "Notes" section of the manual:

"[On Linux], when malloc returns non-NULL there is no guarantee that memory is really available. **If the system is out of memory, one or more processes will be killed by the OOM killer.**"

Running out of Memory

- If you're ever unsure of malloc / free's behavior:

```
$ man malloc
```

- According to the C standard:

"The malloc() function returns a pointer to the allocated memory that is suitably aligned for any kind of variable. **On error, this function returns NULL.**"

You should check for NULL after every malloc():

```
struct student *bob = NULL;
bob = malloc(sizeof(struct student));

if (bob == NULL) {
      /* Handle this.  Often, print and exit. */
}
```